Docket No. 42390.P18124
Express Mail No. EV339917352US

UNITED STATES PATENT APPLICATION

FOR

PROGRAM PHASE DETECTION FOR DYNAMIC OPTIMIZATION

Inventors:

Ara V. Nefian
Ali-Reza Adl-Tabatabai

Prepared by:
BLAKELY SOKOLOFF TAYLOR & ZAFMAN LLP
12400 Wilshire Boulevard, Seventh Floor
Los Angeles, California 90025
(310) 207-3800

PROGRAM PHASE DETECTION FOR DYNAMIC OPTIMIZATION

Background

Field

[0001] The embodiments relate to managed runtime computer system environment technology, and more particularly to dynamic optimization through phase detection.

Description of the Related Art

[0002] Computer programs that are designed to run on managed runtime environments (MRTEs) are distributed in a neutral bytecode format and must be compiled to native machine code by a dynamic compiler. The performance of managed applications depends on the quality of optimization and code generation performed by a compiler. System utilization monitoring can be used to determine when various applications may need to be optimized. As the number of applications running on a system increases, the need for application optimization increases as well.

[0003] Many microprocessor architectures rely on compiler optimizations for performance. Some architectures rely heavily on expensive and sophisticated code-generation optimizations (such as global scheduling and control speculation) for performance.

[0004] In order to optimize executable code, performance feedback and optimization techniques are used. The problem with these techniques is that they are usually intended for hardware implementations or are ad hoc, and thus not suitable for dynamic optimization or software implementations. Moreover, many optimizations require a wait-and-see approach as different optimization criteria are experimented with to achieve optimization. This can be time consuming and may only optimize an application for a short time due to system usage change.

BRIEF DESCRIPTION OF THE DRAWINGS

[0005] The embodiments of the invention are illustrated by way of example and not by way of limitation in the figures of the accompanying drawings in which like references indicate similar elements. It should be noted that references to "an" embodiment of the invention in this disclosure are not necessarily to the same embodiment, and they mean at least one.

[0006] **Figure 1** illustrates one embodiment of a process to detect stable program phases.

[0007] **Figure 2** illustrates a graph of an example buffer of branch trace buffers (BTrB) sample addresses over time.

[0008] **Figure 3** illustrates the histograms corresponding to two phases detected.

[0009] **Figure 4** illustrates the sequence of phases detected when using the data in **Figure 2**.

[0010] **Figure 5** illustrates an embodiment of a system.

[0011] **Figure 6** illustrates an embodiment of a system coupled to another system.

DETAILED DESCRIPTION

[0012] The Embodiments discussed herein generally relate to a method and system for dynamically detecting stable process phases. Referring to the figures, exemplary embodiments will now be described. The exemplary embodiments are provided to illustrate the embodiments and should not be construed as limiting the scope of the embodiments.

[0013] Systems that have dynamic profile guided optimizations (e.g., managed runtime environments, dynamic binary optimizers, and dynamic binary translators) try to determine when to dynamically re-optimize an executing program. **Figure 1** illustrates one embodiment of a process to detect stable program phases for use in dynamic optimization of executable code. Process 100 begins at block 110 with selecting of a phase threshold value. The phase threshold value can be a function of a number of *M* consecutive samples of branch addresses sampled at a time *t*. In one embodiment a user selects the phase threshold value and enters the value as predetermined static parameters in a process. The phase threshold values can also be dynamically modified through a user input device as well.

[0014] Process 100 continues with block 120. In block 120, a number of sequenced buffers are received. In one embodiment, a performance-monitoring unit (PMU) collects the sequenced branch trace buffers (BTrB). The sequenced buffers can be stored in local memory or in files. The buffers received include addresses of the last *L* branches taken. The value of *L* can be predetermined or selected by a user (e.g., 4, 8, 10, etc.). The buffers of the addresses of the branches taken are for a particular sampling moment in time. **Figure 2** illustrates a graph of an example buffer of BTrB sample addresses over time during execution of an example program, such as a benchmarking program.

[0015] After block 120 is complete process 100 continues with block 130. Block 130 determines a distance between centers of at least two consecutive histogram bins. In one embodiment a vector of branch addresses are determined as follows: $b_t=(b_{t,1},\ldots b_{t,L})^T$ is a vector of branch addresses representing a single BTrB sample at time t. $B_t=b_t, b_{t+1,}\ldots b_{tM}$ is a buffer of *M*

consecutive samples made available at one moment of time. M is either predetermined or dynamically adjusted by a user, e.g., 1000, 1400, 1820, etc. A stable phase is defined as a one-dimensional histogram of B_t, and denoted as $H_t=[h_{t,1},\ldots h_{t,N}]^T$. The histogram H_t is a vector of size N where N is the total number of histogram bins. $W_1,\ldots W_N$ is a set of equally spaced and non-overlapping histogram bins that cover the entire space of possible branch addresses. $\Delta W=W_k-W_{k-1}$ is the distance between the centers of two consecutive histogram bins. In one embodiment, a Euclidian distance calculation is used to measure distance, i.e. distance $(\mathbf{H_k}, \mathbf{H_l})= [\sum_{i=1}^{N} (\mathbf{h_{k,i}}-\mathbf{h_{l,i}})^2]^{0.5}$. It should be noted that other distance calculations known in the art can be used as well without deviating from the scope of the embodiments.

[0016] After block 130 has completed, block 140 compares the determined distance with the phase threshold value. If the distance between the two consecutive histogram bins is equal to or larger than the phase threshold value, then the samples in B_k and B_l belong to different phases, otherwise the samples belong to the same phase. Therefore, major execution phases of an executable process are determined based on the comparison result.

[0017] After block 140 is completed, process 100 continues with block 150 if the samples in B_k and B_l belong to the same phase. In one embodiment a variable indicating same phase is set. If the samples in B_k and B_l belong to the different phases, in one embodiment block 145 sets a variable indicating different phases. In one embodiment, block 160 transmits a signal to re-optimize an executing process. The signal can be transmitted, for example, to a dynamic compiler.

[0018] It should be noted that increasing the distance width of the histogram bins ΔW coarsens the resolution and decreases the complexity of phase detection process 100. A coarse resolution is used for phase detection while a fine resolution is used for hot trace detection. Setting $\Delta W = 1$ places every single branch address in a separate histogram bin. This creates a fine-grained histogram. The result of creating a fine-grained histogram is that phase detection process 100 slows down and potentially increases the number of

phases. Setting $\Delta W >> 1$ places branch addresses that are in the same memory region into the same histogram bin. This results in creating a coarse-grained histogram. Creating a coarse grain histogram speeds up phase detection process 100 and reduces the number of phases. By varying the ΔW an analysis of the histograms at different resolutions can be made. Therefore a dynamic trade off of phase detection overhead with phase detection precision can be accomplished. In one embodiment process 100's determination of major execution phases is a dynamic process performed at a predetermined periodic rate. For example, process 100 can be performed at a chosen rate, such as every 5 minutes, hour, 24 hours, etc. In another embodiment, process 100 is manually performed as selected by a user.

[0019] For example purposes, the graph illustrated in **Figure 2** of an example buffer of BTrB sample addresses over time during execution of an example program had the following settings: L=4, M=1820, $\Delta W = 10^5$, and phase threshold = 0.4M. **Figure 3** illustrates the histograms corresponding to two phases detected and **Figure 4** illustrates the sequence of phases detected when using the data in **Figure 2** for 37 blocks of data.

[0020] Process 100 can be used in systems that make use of dynamic profile guided optimizations, such as MRTEs, dynamic binary optimizers, and dynamic binary translators. These types of systems contain hardware performance monitoring and rely on profile-guided optimizations for performance.

[0021] **Figure 5** illustrates an embodiment of a system. System 500 includes processor 510 connected to memory 520. In one embodiment memory 520 is a main memory, such as random-access memory (RAM), static random access memory (SRAM), dynamic random access memory (DRAM), synchronous DRAM (SDRAM), read-only memory (ROM), etc. In another embodiment, memory 520 is a cache memory. Process 100 is in the form of an executable process running in processor 510 and communicating with memory 520. In this embodiment process 100 is a phase detector process that determines major execution phases of another executable process running in memory 520. In system 500, process 100 determines when to re-optimize the other executable process running in system 500. System 500 can be combined

with other known elements depending on the implementation. For example, if system 500 is used in a multiprocessor system, other known elements typical of multiprocessor systems would be coupled to system 500. System 500 can be used in a variety of implementations, such as personal computers (PCs), personal desk assistants (PDAs), notebook computers, servers, MRTEs, dynamic binary optimizers, dynamic binary translators, etc. In one embodiment, the phase detector process 100 exists as a hardware unit having logic and a receiver to receive buffers. The logic elements of the phase detector include circuitry to perform the instructions that process 100 performs, as described above.

[0022] **Figure 6** illustrates a system 600 coupled to a system 650. System 600 includes at least one processor 610, a compiler 620 and a memory 640. Depending on implementation, system 600 also includes other known elements of typical systems, such as multiprocessor systems, PCs, PDAs, etc. Compiler 620 is a dynamic compiler. System 650 includes processor 670, memory 680 and compiler 680. Compilers 620 and 680 are dynamic compilers in one embodiment. In one embodiment process 100 is in the form of an executable process running in system 600 or 650. In this embodiment process 100 is a phase detector process that determines major execution phases of another executable process running in system600 and/or 650. That is, process 100 can run in system 600 to detect major execution phases in system 650, and/or process 100 can run in system 650 to detect major execution phases in system 600. Systems 600 and 650 can be co-located, distributed systems, or networked systems. In one embodiment memory 640 and memory 680 can each be a main memory (such as random-access memory (RAM), static random access memory (SRAM), dynamic random access memory (DRAM), synchronous DRAM (SDRAM), read-only memory (ROM), etc.) or cache memory. In one embodiment, a phase detector is a hardware unit having logic and a receiver to receive buffers. The logic elements of the phase detector include circuitry to perform the instructions that process 100 performs, as described above.

[0023] The above embodiments can also be stored on a device or machine-readable medium and be read by a machine to perform instructions. The machine-readable medium includes any mechanism that provides (i.e.,

stores and/or transmits) information in a form readable by a machine (e.g., a computer). For example, a machine-readable medium includes read-only memory (ROM); random-access memory (RAM); magnetic disk storage media; optical storage media; flash memory devices; biological electrical, mechanical systems; electrical, optical, acoustical or other form of propagated signals (e.g., carrier waves, infrared signals, digital signals, etc.). The device or machine-readable medium may include a micro-electromechanical system (MEMS), nanotechnology devices, organic, holographic, solid-state memory device and/or a rotating magnetic or optical disk. The device or machine-readable medium may be distributed when partitions of instructions have been separated into different machines, such as across an interconnection of computers.

[0024] While certain exemplary embodiments have been described and shown in the accompanying drawings, it is to be understood that such embodiments are merely illustrative of and not restrictive on the broad invention, and that this invention not be limited to the specific constructions and arrangements shown and described, since various other modifications may occur to those ordinarily skilled in the art.